CEWES MSRC/PET TR/99-11

A Computation Allocation Model for Distributed Computing Under MPI_Connect

by

Clay P. Breshears
Graham Fagg

04h01099

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
Nichols Research Corporation

A Computation Allocation Model for Distributed Computing Under MPI_Connect

Clay P. Breshears[*] Graham Fagg[†]

April 15, 1999

1 Introduction

Parallel and distributed computing techniques have been used effectively to speed up the execution times of a multitude of applications. How parallel code is written depends on many factors related not only to the application under consideration, but also on the hardware and supporting software available. A programmer undertaking the task of writing a parallel code faces several choices:

- What is the underlying programming model for the target execution platform (distributed- or shared-memory)?
- How should the division of work within the computation be accomplished (task- or data-parallel)?
- What amount of work should be distributed to individual processors (coarse- or fine-grain)?

Another fundamental concern when writing parallel codes is balancing the amount of work done by each processor. Since the overall execution time often is dependent upon the time taken by the processor running the task with the longest execution time, it behooves the programmer to ensure that an equitable distribution of work is assigned to each processor within the computation. Some applications have a natural, static division of equivalent amounts of work while others have a more unpredictable *a priori* partition of labor, often related directly to the input data. A dynamic load balancing method to assign computations in a way that attempts to have all processors finish their work at the same time would be best suited for this latter set of applications. One such dynamic task allocation method is known as the Master-Worker model.

We present a modified version of the Master-Worker model, called the Queen-Drone model, that is adapted to run coarse-grain, task-parallel computations on collections of distributed high performance computing (HPC) platforms under the MPI_Connect intercommunication library. This name was chosen for the analogy of a single queen bee assigning tasks to drone bees; disjoint groups of drones are collected into hives which correspond to groups of parallel worker processes external to the queen process, typically executing on separate parallel machines. We also present some experiences and execution results of our model applied to a simple integer factoring code and a

[*]Rice University On-site Scalable Parallel Programming Tools Lead for PET
[†]University of Tennessee, Knoxville, TN

large-scale harbor wave response code. We have been able to run our codes on separate machines of the same architecture, as well as between machines of different manufacture.

Though care was taken to remove language specific details, the reader should be aware that, for this report, all codes discussed were written in Fortran and the Fortran names of routines are given in the text. The authors do not foresee any difficulty implementing the models or algorithms discussed with the C language. Section 2 reviews the Master-Worker computation model while Section 3 reviews MPI_Connect. The Queen-Drone model is presented and discussed in Section 4. Our experiences with two parallel application codes are described in Section 5. Section 6 presents some conclusions and future plans for use of the Queen-Drone algorithm.

2 Master-Worker Model

The Master-Worker is a classic parallel programming model, and various citations in the literature are available; e.g., [5, 3]. One process (the master, typically executing on a single processor) is devoted to assigning computations to all the other processes (the workers, each executing on separate processors) involved. When a worker process completes a task, a request for more work is sent to the master process, which replies with one of two messages: a message containing the next task to be computed, or, if all work has been assigned, a message signaling that the parallel computation has been completed. Upon receipt of the termination signal from the master, worker processes exit gracefully. In the case where there is some serial computation to be performed before resuming parallel execution, the master process may detain workers by not immediately replying to worker requests for more tasks until the application warrants it. Algorithms 1 and 2 display pseudo-code for the master and worker processes.

Depending upon the application requirements, it is possible that the master process will perform some pre-processing of data in order to prepare a task for assignment to a worker. However, more often than not, the master process merely reads task data from a file and relays this to workers. The master process remains mostly idle during the parallel execution of tasks. The appeal of the Master-Worker model is in the immediate dispatch of a new task in response to worker requests which reduces worker idle time.

To execute this programming model, the use of a message passing library (e.g., MPI) is essential. While the Master-Worker model is easily programmed to run on a single parallel machine, running such a model for a single application across distributed HPC machines, each running a potentially different version of MPI, presents interesting challenges. In order to meet those challenges, we developed a variation of the Master-Worker model known as the Queen-Drone model. We have programmed this model using MPI_Connect.

3 MPI_Connect

MPI_Connect [4] is a metacomputing middleware–executable from either C or Fortran codes–that allows separately initiated MPI applications to interact in a peer-peer fashion. It allows dynamic connect and disconnect of disjoint collections of processors during the execution of applications. The MPI 1.1 standard [7] only supports a static process model; i.e., all processors that would participate in a computation are known at the outset of execution.

While MPI_Connect was originally built upon MPICH [6] and LAM6 [2], it has been extended to

Algorithm 1 Master Process Pseudo-code

```
do i = 1 to number_of_tasks
  blocking receive      ! wait for work request
  blocking send         ! send task data
end do
      ! All tasks have been assigned
do i = 1 to num_procs - 1
  blocking receive      ! wait for work request
  blocking send         ! send termination signal
end do
```

Algorithm 2 Worker Process Pseudo-code

```
do infinite loop
  blocking send         ! ask master for work
  blocking receive      ! get task data
  if not termination signal then
    perform calculations
  else
    exit infinite loop
  end if
end do
```

support IBM MPIF and SGI MPI. One of the strengths of MPI_Connect is the capability to have the different applications running under different versions of MPI. This is true whether the applications are running on the same or different platforms. Thus the user is able to utilize the optimized vendor version of MPI on each HPC platform involved in the computations. MPI_Connect currently uses PVM as the interconnection and process management layer.

Under MPI_Connect, MPI codes interoperate by naming themselves with a third party naming service that coordinates the interaction. This naming service and its location are transparent to the code. The names used are plain ASCII and are completely freeform to aid debugging. Once MPI applications have named themselves they can then locate other MPI applications by using the naming service. When both parties agree, an MPI intercommunicator is created between them and this is then used in the same way as any other intercommunicator in MPI. In order to disconnect from all intercommunicators established with other codes, the MPI application removes its name from the naming service.

Configuring current MPI applications to run under MPI_Connect is accomplished with a minimal impact to the code. Three extra commands are added:

- `MPI_CONN_REGISTER()` to register the code's name with the naming service,
- `MPI_CONN_INTERCOMM_CREATE()` to create an intercommunicator with another named application, and
- `MPI_CONN_LEAVE()` to remove the code's name from the naming service and consequently sever all MPI_Connect intercommunicators in which the code is involved.

Multiple calls to `MPI_CONN_INTERCOMM_CREATE()` may be needed, one for each application for which intercommunication is to be established.

Using the intercommunicator returned from the `MPI_CONN_INTERCOMM_CREATE()` call in the appropriate MPI communication functions in order to refer to external applications is the only other change needed. Upon execution of a MPI call, the MPI_Connect library tests the communicator used. If the communicator is local to the application, the local profiling library version of the MPI call involved is executed. If the communicator was returned from a previous call to `MPI_CONN_INTERCOMM_CREATE()`, the appropriate actions to complete the MPI function are handled by the MPI_Connect intercommunication library and process management software. This process is transparent to the user.

4 Queen-Drone Model

The Master-Worker model was originally designed and intended to run on a single HPC platform with all processors able to communicate through some message passing mechanism. By running on a single platform, applications load-balanced through the Master-Worker algorithm are restricted to the total number of processors available on that machine. In some cases the scalability of the application is such that a relatively small number of processors is adequate to achieve a high level of efficiency. Other applications, especially in the realm of scientific computation and simulation, can make effective use of hundreds or even thousands of processors. Since very few commercially available HPC systems are equipped with thousands of processors, a user must coordinate an application running across multiple machines to harness the computational power of a very large number of processors. Under PVM, it is possible to run the Master-Worker model across multiple platforms. However, since MPI has become the *de facto* standard for message passing, we have chosen to develop and study codes under MPI.

Our model for coordination of many different processors executing tasks on different HPC platforms is based on the analogy of a queen bee directing the work of many drone bees, which are grouped into disjoint hives. The queen represents the master process parceling out work; the drone bees represent the worker processes doing the computations assigned to them; the hives are separate HPC systems on which the drones carry out their tasks. The Queen-Drone model is a modification of the Master-Worker model which utilizes MPI and MPI_Connect as the underlying communication fabric between the queen process and the drone processes executing on different machines. Algortithms 3 and 4 show pseudo-code to implement the queen process and the drone process.

For the remainder of the paper, we make the assumption that the queen is the only process within a communicator with all drones collected into other, disjoint communicators. While it is possible to run the queen and drones within the same communicator, this would require special handling of “local” communication between queen and resident drones. Such a special case, while not overly difficult to program, is beyond the focus of this paper.

4.1 Queen-Drone Algorithm Details

The first thing that each drone process does is to register the hive name and create an intercommunicator with the queen process. Each drone process must call `MPI_CONN_REGISTER()` to declare its participation within the hive communicator and `MPI_CONN_INTERCOMM_CREATE()` in order to get a handle for the queen’s communicator. This handle is used in `MPI_SEND()` and `MPI_RECV()` routine calls as the communicator parameter. Before any drone in the hive requests any tasks, one drone

Algorithm 3 Queen Process Pseudo-code

```
connect with hives          ! intercomm set up
do i = 1 to number_of_tasks
   probe for drone request          ! busy wait on comm
   blocking receive
   blocking send          ! send task data
end do
         ! All tasks have been assigned
do worker = 1 to num_drones
   probe for drone request          ! busy wait on comm
   blocking receive
   blocking send          ! send termination signal
end do
```

Algorithm 4 Drone Process Pseudo-code

```
connect with queen          ! intercomm set up
do infinite loop
   blocking send          ! ask queen for work
   blocking receive          ! get task data
   if not termination signal then
      perform calculations
   else
      exit infinite loop
   end if
end do
```

process (designated as the *lead drone*) sends an informational message to the queen containing the total number of drones that are active in the hive. The queen adds this count to a total of active drones and responds by returning a unique *hive number* to the lead drone. The hive number is broadcast to the other drone processes in the hive. In this way, all drones within a hive are synchronized to a point in the execution at which it is known the queen has created all hive intercommunicators and is ready to handle task requests.

Informational messages from the lead drones contain a single integer count of the number of drones in the hive. Task request messages from drones to the queen process need not contain any data; the tag is sufficient to denote that the message is a request for more work. Similarly, messages from the queen to drone processes use the tag to denote whether the message carries task data or the message is the termination signal. As an aid for debugging during the development of the Queen-Drone algorithm, we encoded the hive number and the sending drone's rank within the hive communicator into a single integer and used this value as the data portion of the task request messages sent to the queen process.

The first thing that the queen process must do is register her name and create the intercommunicators between herself and the drone processes. One communicator per hive is sufficient since all drones within the hive are part of the same MPI_Connect communicator and have a unique rank within that communicator. Intercommunicators that are returned from each call to `MPI_CONN_INTERCOMM_CREATE()` are kept in an array of intercommunicators within the queen process.

The MPI standard allows the use of wildcard placeholders to match on any sender and/or any message tag within the `MPI_RECV()` routine. There is no wildcard for the communicator argument in the parameter list. Thus, to execute the "`blocking receive`" steps within the queen process, the explicit communicator from which the message is to be received must be known. This presents no problem in the Master-Worker model since all processes are within the same communicator. However, since drones are in communicators different than the queen and there may be two or more hive intercommunicators, the queen process must "`probe for drone request`."

To implement this "`probe for drone request`" step, the queen continuously loops over each intercommunicator within her array, in turn. Each intercommunicator is used as an argument to `MPI_IPROBE()` with a sender wildcard placeholder. This MPI routine immediately returns a TRUE value if a message has arrived from the stated sender within the stated intercommunicator, or FALSE if no such message is awaiting receipt. By using the sender wildcard as an argument to `MPI_IPROBE()`, the queen probes for the arrival of any message from the given intercommunicator. When the probe function returns TRUE, the probe loop is exited and it is known which explicit intercommunicator to use for the `MPI_RECV()` routine call that follows.

A case-selection structure to handle both task requests and informational messages is easily implemented. This technique hides latency of the informational messages between the queen and drone processes. The queen process is able to assign work to drones before all informational messages (which may be delayed due to network traffic) from other hives have been received.

4.2 Queen-Drone Advantages

We have identified several advantages of running codes under the Queen-Drone model including a minimal amount of code changes to implement from a Master-Worker code and the capability to run on multiple HPC platforms. Each of these is discussed in more detail below.

4.2.1 Minimal Code Changes

The changes needed to convert a current Master-Worker MPI code to an equivalent Queen-Drone code are minimal. Perhaps the most challenging part of the conversion is to split the code into two separate programs: one for the queen process and the other for the drones. Both codes require the addition of MPI_Connect registration and departure routine calls. Besides the two drone request probe loops, the queen program also requires code to process the names of the participating hives in order to establish intercommunicators with each and to handle the informational messages from the lead drones. If a static set of hives is to be used each time the codes are run, all hive names could be incorporated directly into the program. However, if a dynamic set of hives is desired, a hive name file, listing names of all participating hives and updated before the queen begins execution, is used. Unique hive names are best included within each copy of the drone code.

The drone program requires a call to the `MPI_CONN_INTERCOMM_CREATE()` routine that is executed by each drone. Code to choose a lead drone in each hive (the process with rank zero is easiest to elect leader) and coordinate the receipt and broadcast of the confirmation from the queen process must also be included.

4.2.2 Multiple HPC Platforms

Applications under the Queen-Drone model may be run on multiple, geographically separated HPC platforms. The queen process may also be running on a single HPC processor. However, since the queen remains mostly idle during the bulk of the execution, it is possible to run the queen process on a workstation if no specialized HPC services are required to prepare tasks for distribution to drone processes. In this way, no expensive HPC resources are used to execute a process that does little computation.

Each HPC system can be running one or more copies of the drone program and each group of drones will execute as if it were the only hive participating in the computation. This allows the user the flexibility to compute with as many or as few drones distributed among as many or as few hives as fit the requirements of the application or physical resources available. The independence of hives allows a dynamic allocation of resources from one run to the next. That is, while the static allocation of the number of drones and hives must be fixed at the outset of execution, subsequent computations need not be required to use the same configuration as a previous execution.

4.2.3 Best Platform for Tasks

If a computation is made up of homogeneous tasks, the choice of HPC platforms to be used is almost irrelevant. However, if a computation contains heterogeneous tasks that are better suited for different HPC systems, the queen process can be programmed to assign tasks to processors on appropriate architectures. Unless a fixed assignment configuration of hives to HPC resources is programmed into the code, the informational message to the queen process is used to identify a hive's execution platform.

To correctly assign tasks to drones on specific machines, two methods are available to the queen. The queen process screens for specific communicators from which to receive task requests for the assignment of the next task if the order of task assignment is critical. On the other hand, if there is no dependence between the execution order of different types of heterogeneous tasks, multiple queues of tasks are kept. When a task request is received, the platform running the requesting task is determined and the task at the head of the proper queue is assigned. Once a queue for a HPC platform becomes empty, drones executing on the corresponding machine type may be terminated or assigned tasks from other queues still containing work, if appropriate.

5 Applications

In this section we present some of our experiences in using the Queen-Drone model to program two parallel applications. The first is a simple integer factoring code that was used to develop and debug the Queen-Drone algorithm. The second, CGWAVE, is a harbor wave response code developed by researchers at the Coastal and Hydraulics Laboratory at U. S. Army Corps of Engineers Waterways Experiment Station (CEWES).

5.1 Integer Factoring

To demonstrate the viability of the Queen-Drone model, a simple application with tasks of differing execution times was sought. We decided on an integer factoring algorithm. The factoring is done by the brute force method of dividing the number under consideration by all integers between two (2) and the square root of the original until all prime factors are found. The number of factors and number of potential factors that must be tested in such an algorithm guarantees a wide range of execution times for different inputs. However, factoring a single number is still a relatively fine-grained task. To create more coarse-grained tasks, the factoring algorithm was embedded within a loop that factored, in turn, all numbers between the initial input argument and twice that argument.

The task data for each drone process is simply the integer argument which is doubled by the drone to determine the range of numbers to be factored. We were able to run the drone processes on two separate SGI/Cray Origin2000 systems. These platforms are located at the CEWES Major Shared Resource Center (MSRC) in Vicksburg, MS, and the Aeronautical Systems Center (ASC) MSRC at Wright-Patterson Air Force Base in Dayton, OH. We were able to execute the code with both Origin2000 systems simultaneously involved and more than one hive allocated to a single Origin. During separate executions, the queen process was successfully run on both one processor of the CEWES MSRC Origin as well as a Sun workstation located at the University of Tennessee, Knoxville.

5.2 CGWAVE

CGWAVE is a Fortran 77 code used for military and civil engineering applications. The code simulates the response of a harbor's surface to the stimulus of waves from the open ocean within the bathymetry of the harbor. The underlying mathematical model is a two-dimensional, elliptic mild-slope wave equation which leads to a Helmholtz-type equation. The resulting large, sparse system is solved via conjugate gradient. Details of this code can be found in [9].

To use the code effectively, a number of different open ocean conditions are modeled. The set of parameters describing the waves entering the harbor (amplitude, periodicity, direction) is known as a wave component. Each wave component yields an independent sparse system of equations, the solution of which is combined with the solution of all other wave components. Thus, CGWAVE is able to distribute tasks of wave components to drones with a post-processing step added to gather the results from all computations to form the final harbor response solution.

Because the execution time for individual wave components can often be measured in hours, running a small number (e.g., 75) of wave components can take weeks of CPU time. Overall execution time can easily be reduced by running wave components in parallel. For example, on 32 processors (a not unlikely number of processors to be found on commercially available HPC platforms), 75 components could take about 24 hours to complete. However, researchers are interested in computing solutions that involve 500 to 1000 wave components. Even on 32 processors, this would take weeks to compute in the worst case. Since all wave component computations are independent from one another, more processors can be employed to reduce the wallclock execution time.

The serial CGWAVE code was first parallelized with a Master-Worker model by researchers at CEWES MSRC. After our success with the factoring application described above, the Queen-Drone algorithm was incorporated into CGWAVE. We have run these queen and drone processes on SGI/Cray Origin2000 systems located at CEWES MSRC and ASC MSRC. As expected, the execution time for a given suite of wave components depends on the total number of processors able to be assigned to the computations.

As another facet in the parallelization of CGWAVE, OpenMP [8] directives were used to parallelize the conjugate gradient solver for individual wave components. OpenMP threads are spawned on separate processors to compute certain loop iterations in parallel. This modification further reduces execution time of wave components as long as extra processors are available. Even though only a small number of processors may be available on any one HPC platform, multiple platforms can be connected under the Queen-Drone implementation to access potentially hundreds of processors for execution of OpenMP threads. Further details of CGWAVE implemented under this dual-level parallelism may be found in [1].

6 Conclusions and Future Work

We have presented the Queen-Drone model of parallel task allocation under MPI_Connect. To demonstrate the effectiveness of this model for coarse-grain, task-parallel computations, two parallel applications have been coded and successfully run with the Queen-Drone algorithm used to distribute tasks across multiple HPC platforms.

Future work on the CGWAVE code will include taking advantage of being able to assign tasks to different HPC platforms that are best suited to execute those tasks under the Queen-Drone model. The computation time for wave components has been found to be dependent upon the amplitude and length of the wave period under consideration. With this in mind, it would be possible to assign short period wave components to processors on one parallel machine (e.g., IBM SP) while the long period components could be assigned to processors on an SGI/Cray Origin2000 which would be able to compute with OpenMP threads. In fact, along with the wave component data, the number of threads to be spawned in the execution could also be sent to a drone process on an Origin2000. This heterogeneous combination of HPC platforms would be useful in cases where there is a severe limitation on the resources of one or more platforms; e.g., only a handful of Origin2000 processors were free while, at the same time, a small number of SP processors could be used. The ease and versatility of programming achievable with MPI_Connect and the Queen-Drone model make such cross-architecture computations possible.

References

[1] S. W. Bova, C. P. Breshears, C. Cuicchi, Z. Demirbilek, and H. A. Gabb, "Dual-level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP", CEWES MSRC/PET Technical Report, *to be published*, 1999.

[2] G. D. Burns, R. B. Daoud, and J. R. Vaigl, "LAM: An Open Cluster Environment for MPI", *Proceedings of the Supercomputing Symposium '94*, Toronto, Canada, June 1994.

[3] H. El-Rewini and T. G. Lewis, *Distributed and Parallel Computing*, Manning Publications Co., Greenwich, CT, 1998.

[4] G. E. Fagg and J. J. Dongarra, *PVMPI: An Integration of the PVM and MPI Systems*, Technical Report UT-CS-96-328, University of Tennessee, Knoxville, May 1996.

[5] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley Publishing Company, 1995.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", *Parallel Computing* **22**:789–828, 1996.

[7] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard (version 1.1)*, Technical Report, 1995, `http://www.mpi-forum.org`.

[8] OpenMP Architecture Review Board, *OpenMP Fortran Application Program Interface*, Version 1.0, October 1997, `http://www.openmp.org`.

[9] B. Xu, V. Panchang, and Z. Demirbilek "Exterior Reflections in Elliptic Harbor Wave Models", *Journal of Waterway, Port, Coastal, and Ocean Engineering* **212**:118–126, May/June 1996.